A Generic Library for Floating-Point Numbers and Its Application to Exact Computing

Marc Daumas, Laurence Rideau, Laurent Thery

A Generic Library for Floating-Point Numbers and Its Application to Exact Computing

Marc Daumas[1], Laurence Rideau[2], and Laurent Théry[2]

[1] CNRS, Laboratoire de l'Informatique du Parallélisme
UMR 5668 - ENS de Lyon - INRIA
Marc.Daumas@ens-lyon.fr
[2] INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis France
{Laurence.Rideau,Laurent.Thery}@sophia.inria.fr

Abstract. In this paper we present a general library to reason about floating-point numbers within the Coq system. Most of the results of the library are proved for an arbitrary floating-point format and an arbitrary base. A special emphasis has been put on proving properties for exact computing, i.e. computing without rounding errors.

1 Introduction

Building a reusable library for a prover is not an easy task. The library should be carefully designed in order to give direct access to all key properties. This work is usually underestimated. Often libraries are developed for a given application, so they tend to be incomplete and too specific. This makes their reuse problematic. Still we believe that the situation of proving is similar to the one of programming. The fact that the programming language Java was distributed with a quite complete set of libraries has been an important factor to its success.

This paper presents a library for reasoning about floating-point numbers within the Coq system [18]. There has already been several attempts to formalize floating-point numbers in other provers. Barrett [2] proposed a formalization of floating-point numbers using the specification language Z [33]. Miner [25] was the first to provide a proving environment for reasoning about floating-point numbers. It was done for PVS [30]. More recently Harrison [16] and Russinoff [31] have developed libraries for HOL [14] and ACL2 [21] respectively and applied them successfully to prove the correctness of some algorithms and hardware designs. When developing our library we have tried to take a generic approach. The base of the representation and the actual format of the mantissa and of the exponent are parameterized. We still use the key property of correct rounding and the clean ideas of the IEEE 754 and 854 standards [35,7,13].

The paper is organized as follows. We first present some basic notions and their representation in Coq. Then we spend some time to explain how we have defined the central notion of rounding. In Section 4, we give examples of the kind of properties that are in the library and how they have been proved. Section 5 details the proof of correctness of a program that is capable to detect the

R.J. Boulton and P.B. Jackson (Eds.): TPHOLs 2001, LNCS 2152, pp. 169–184, 2001.

base of an arbitrary arithmetic. Finally we show an application of this library to floating-point expansions. These expansions were presented in [11] first and more formally in [28,32]. The technique can be specialized for the predicates of computational geometry [5,19] or to small-width multiple precision arithmetic [1,9], among other applications.

2 Floating-Point Format and Basic Notions

2.1 Definitions

Our floating-point numbers are defined as a new type composed by records:

Record. *float*: *Set* := *Float* {*Fnum*:$\mathbb{Z}$; *Fexp*:$\mathbb{Z}$}

This command creates a new type *float*, a constructor function *Float* of type $\mathbb{Z} \to \mathbb{Z} \to$ *float* and two destructor functions *Fnum* and *Fexp* of type *float* $\to \mathbb{Z}$. The fact that *float* is of type *Set* indicates that *float* is a datatype. The component *Fnum* represents the mantissa and *Fexp* the exponent. In the following we write (*Float* x y) simply as $\langle x, y\rangle$ and (*Fnum* p) and (*Fexp* p) as $n[p]$ and $e[p]$ respectively.

In order to give a semantics to this new type, we have to relate our *float* to their value as a real. The reals in Coq are defined axiomatically [24] as the smallest complete archimedian field. We define the function *FtoR* of type *float* $\to$ $\mathbb{R}$ as:

Definition. $FtoR := \lambda p\text{: }float.\, n[p] * \beta^{e[p]}$.

This definition is parameterized over an arbitrary base β. We suppose that the base is an integer strictly greater than one. Our notation differs from the IEEE standard notation [35] and even from the pre-standard notation [6]. The mantissa is an integer and $\beta^{e[x]}$ is one unit in the last place of the float x or the weight of its least significant bit.

Having the type *float* as a separate type instead of a subtype of the real numbers as in [16] implies that we have to manipulate two notions of equality. The usual Leibniz equality $p = q$ means that p and q have the same components as a record. The equality over $\mathbb{R}$ (*FtoR* p) = (*FtoR* q) means that they represent the same real. In the following this last equality will be denoted $p == q$ and the function *FtoR* is used implicitly as a coercion between our floating-point numbers and the reals, so $0 < p$ should be understood as $0 <$ (*FtoR* p). The two notions of equality are related. For example we have the following theorems:

Theorem. $FtoREqInv_1 : \forall p, q\text{: }float.\, \neg p == 0 \Rightarrow p == q \Rightarrow n[p] = n[q] \Rightarrow p = q$.

Theorem. $FtoREqInv_2 : \forall p, q\text{: }float.\, p == q \Rightarrow e[p] = e[q] \Rightarrow p = q$.

On the type *float*, we can define the usual operations that return an element of type *float* such as:

Definition. $Fplus := \lambda p, q\text{: }float.$
$\langle n[p] * (\beta^{e[p]-min(e[p],e[q])}) + n[q] * (\beta^{e[q]-min(e[p],e[q])}), min(e[p], e[q])\rangle$

Definition. $Fop := \lambda p\colon float.\ \langle -n[p], e[p]\rangle$.

Definition. $Fabs := \lambda p\colon float.\ \langle |n[p]|, e[p]\rangle$.

Definition. $Fminus := \lambda p, q\colon float.\ (Fplus\ p\ (Fop\ q))$.

Definition. $Fmult := \lambda p, q\colon float.\ \langle n[p] * n[q], e[p] + e[q]\rangle$.

For each of these functions we have proved a theorem of correctness. For addition this theorem looks like:

Theorem. $Fplus\_correct\colon \forall p, q\colon float.\ (Fplus\ p\ q) == p + q$.

where the rightmost addition is the usual addition on real numbers. Note that since we do not have uniqueness of representation, these functions just pick a possible representant of the result. In the following we write $+,-,|\ |,\ -,*$ for *Fplus*, *Fop*, *Fabs*, *Fminus*, *Fmult* respectively.

2.2 Bounded Floating-Point Numbers

As it is defined, the type *float* contains too many elements. In order to represent machine floating-point numbers we need to define the notion of bound:

Record. *Fbound*: *Set* := *Bound* {*vNum*:$\mathbb{N}$; *dExp*:$\mathbb{N}$}

We use this notion of bound to parameterize our development over an arbitrary bound b. In the following, we write (*vNum* b) and (*dExp* b) as $N[b]$ and $E[b]$. With this arbitrary bound we can define a predicate *Fbounded* to characterize bounded floating-point numbers:

Definition. $Fbounded := \lambda p\colon float.\ -N[b] \leq n[p] \leq N[b] \wedge -E[b] \leq e[p]$.

In the following we write (*Fbounded* p) as $\mathcal{B}[p]$. A real that has a bounded floating-point number equivalent is said to be *representable*. Note that we do not impose any upper bound on the exponent. This allows us to have a more uniform definition of rounding since any real is always between two bounded floating-point numbers.

In existing systems, overflows generate combinatorial quantities like infinities, errors (NaN) and so on. Having the upper bound would force us to treat these non-numerical quantities at each theorem. The bound should rather be added only to the final data type. Only the high level theorems will be proved both for numerical and for combinatorial values.

Removing the lower bound is not admissible as it will hide the difficult question of the subnormal numbers. As can be seen for example in [10], the lower bound is used to prove properties through the full set of floating-point numbers and not uniquely on small numbers.

2.3 Canonical Numbers

So far the bound on the mantissa is arbitrary. In practice, it is set so that any number is represented with a fixed width field. The width of the field is called the *precision*. We define an arbitrary integer variable *precision* that is supposed

not null and we add the following hypothesis:

Hypothesis. $pGivesBound$: $N[b] = \beta^{precision} - 1$.

This insures that the number of digits of the mantissa is at most *precision* in base β. We can also define a notion of *canonical* representant.

We first define the property of a floating-point number to be *normal* if it is bounded and the number of digits of its mantissa is exactly the precision:

Definition. $Fnormal := \lambda p\colon float.\, \mathcal{B}[p] \wedge digit(p) = precision$.

where *digit* is a function that returns the number of radix-β digits in the integer $n[p]$ (no leading zeros). All bounded numbers do not have a normal equivalent, take for example $\langle 0, 0\rangle$. For numbers near zero, we define the property of being *subnormal* by relaxing the constraint on the number of digits:

Definition. $Fsubnormal := \lambda p\colon float.\, \mathcal{B}[p] \wedge e[p] = -E[b] \wedge digit(p) < precision$.

We can now define what it is for a number to be *canonic* as:

Definition. $Fcanonic := \lambda p\colon float.\, (Fnormal\ p) \vee (Fsubnormal\ p)$.

In the following the properties $(Fnormal\ p)$, $(Fsubnormal\ p)$, $(Fcanonic\ p)$ will be denoted as $\mathcal{N}[p]$, $\mathcal{S}[p]$, and $\mathcal{C}[p]$ respectively. It is easy to show that normal, subnormal and canonic representations are unique:

Theorem. $FnormalUnique$: $\forall p, q\colon float.\, \mathcal{N}[p] \Rightarrow \mathcal{N}[q] \Rightarrow p == q \Rightarrow p = q$.

Theorem. $FsubnormalUnique$: $\forall p, q\colon float.\, \mathcal{S}[p] \Rightarrow \mathcal{S}[q] \Rightarrow p == q \Rightarrow p = q$.

Theorem. $FcanonicUnique$: $\forall p, q\colon float.\, \mathcal{C}[p] \Rightarrow \mathcal{C}[q] \Rightarrow p == q \Rightarrow p = q$.

In order to compute the canonical representant of a bounded number, we build the following function:

Definition. $Fnormalize := \lambda p\colon float.$
`if` $n[p] = 0$ `then` $\langle 0, -E[b]\rangle$
`else let` $z = min(precision - digit(p), |E[b] + e[p]|)$ `in` $\langle n[p] * \beta^z, e[p] - z\rangle$.

The following two theorems insure that what we get is the expected function:

Theorem. $FnormalizeCorrect$: $\forall p\colon float.\, (Fnormalize\ p) == p$.

Theorem. $FnormalizeCanonic$: $\forall p\colon float.\, \mathcal{B}[p] \Rightarrow \mathcal{C}[(Fnormalize\ p)]$.

With the function *Fnormalize*, it is possible to capture the usual notion of *unit in the last place* with the following definition:

Definition. $Fulp := \lambda p\colon float.\, \beta^{e[(Fnormalize\ p)]}$.

Working with canonical representations not only do we get that equality is the syntactic one but also the comparison between two numbers can be interpreted directly on their components with lexicographic order on positive numbers:

Theorem. $FcanonicLtPos$: $\forall p, q\colon float.\, \mathcal{C}[p] \Rightarrow \mathcal{C}[q] \Rightarrow$
$0 \le p < q \Rightarrow (e[p] < e[q]) \vee (e[p] = e[q] \wedge n[p] < n[q])$.

We have a similar theorem for negative floating-point numbers. These two the-

orems give us a direct way to construct the successor of a canonical number:

Definition. *FSucc* := λp: *float*.
`if` $n[p] = N[b]$ `then` $\langle \beta^{precision-1}, e[p]+1\rangle$
`else if` $n[p] = -\beta^{precision-1}$ `then`
`if` $e[p] = -E[b]$ `then` $\langle n[p]+1, e[p]\rangle$ `else` $\langle -N[b], e[p]-1\rangle$
`else` $\langle n[p]+1, e[p]\rangle$

To be sure that this function is the expected one, we have proved the three following theorems:

Theorem. *FSuccCanonic*: $\forall p$: *float*. $\mathcal{C}[p] \Rightarrow \mathcal{C}[(FSucc\ p)]$.

Theorem. *FSuccLt*: $\forall p$: *float*. $p < (FSucc\ p)$.

Theorem. *FSuccProp*: $\forall p, q$: *float*. $\mathcal{C}[p] \Rightarrow \mathcal{C}[q] \Rightarrow p < q \Rightarrow (FSucc\ p) \leq q$.

The function *FPred* that computes the preceeding canonical number can also be defined in a similar way.

3 Rounding Mode

Rounding plays a central role in any implementation of floating-point numbers. Following the philosophy of the IEEE standard, all operations on floating-point numbers should return the rounded value of the result of the exact operation. The logic of Coq is constructive: every function definition has to be explicit. In such a context defining a rounding function is problematic. We overcome this problem by defining rounding as a relation between a real number and a floating-point number. Rounding is defined abstractly. The first property a rounding must verify is to be total:

Definition. *TotalP* := λP: $\mathbb{R} \to$ *float* $\to$ *Prop*. $\forall r$: $\mathbb{R}$. $\exists p$: *float*. $(P\ r\ p)$.

In Coq, propositions are of type *Prop*, so an object P of type $\mathbb{R} \to$ *float* $\to$ *Prop* is a relation between a real and a floating-point number. Another property that is needed is the compatibility:

Definition. *CompatibleP* := λP: $\mathbb{R} \to$ *float* $\to$ *Prop*. $\forall r_1, r_2$: $\mathbb{R}$. $\forall p, q$: *float*.
$(P\ r_1\ p) \Rightarrow r_1 = r_2 \Rightarrow p == q \Rightarrow \mathcal{B}[q] \Rightarrow (P\ r_2\ q)$.

Although we defined a canonical representation of floating-point numbers, we will not specify that the rounded value of a floating-point number should be canonical. This is definitively not needed at this point and we will see later that being more general allows us to build easier proofs. We specify that the rounding must be monotone:

Definition. *MonotoneP* := λP: $\mathbb{R} \to$ *float* $\to$ *Prop*. $\forall r_1, r_2$: $\mathbb{R}$. $\forall p, q$: *float*.
$r_1 < r_2 \Rightarrow (P\ r_1\ p) \Rightarrow (P\ r_2\ q) \Rightarrow p \leq q$.

Finally looking for a projection, we set that the rounded value of a real must be one of the two floats that are around it. When the real to be rounded can be represented by a bounded floating-point number, the two floating-point numbers around it are purposely equal. We define the ceil (*isMin*) and the floor (*isMax*)

relations and the property for a rounded value to be either a ceil or a floor:

Definition. $isMin := \lambda r\colon \mathbb{R}.\, \lambda min\colon float.$
$\mathcal{B}[min] \wedge min \leq r \wedge \forall p\colon float.\, \mathcal{B}[p] \Rightarrow p \leq r \Rightarrow p \leq min.$

Definition. $isMax := \lambda r\colon \mathbb{R}.\, \lambda max\colon float.$
$\mathcal{B}[max] \wedge r \leq max \wedge \forall p\colon float.\, \mathcal{B}[p] \Rightarrow r \leq p \Rightarrow max \leq p.$

Definition. $MinOrMaxP := \lambda P\colon \mathbb{R} \to float \to Prop.$
$\forall r\colon \mathbb{R}.\, \forall p\colon float.\, (P\ r\ p) \Rightarrow (isMin\ r\ p) \vee (isMax\ r\ p).$

Using the previous definitions, we can define what is a rounding mode:

Definition. $RoundedModeP := \lambda P\colon \mathbb{R} \to float \to Prop.$
$(TotalP\ P) \wedge (CompatibleP\ P) \wedge (MinOrMaxP\ P) \wedge (Monotone\ P).$

Having defined the rounding abstractly gives us for free the possibility of proving general properties of rounding. An example is the property that the rounding of a bounded floating-point number is the number itself. It can be stated as:

Definition. $ProjectorP := \lambda P\colon \mathbb{R} \to float \to Prop.$
$\forall p, q\colon float.\, \mathcal{B}[p] \Rightarrow (P\ p\ q) \Rightarrow p == q.$

Theorem. $RoundedProjector\colon \forall P\colon \mathbb{R} \to float \to Prop.$
$(RoundedModeP\ P) \Rightarrow (ProjectorP\ P).$

As a matter of fact we could have replaced in the definition of *RoundModeP* the property *MinOrMax* by *ProjectorP*.

We can now define the usual rounding modes. First of all, the two relations *isMin* and *isMax* are rounding:

Theorem. $MinRoundedModeP\colon (RoundedModeP\ isMin).$

Theorem. $MaxRoundedModeP\colon (RoundedModeP\ isMax).$

The rounding to zero is defined as follows:

Definition. $ToZeroP := \lambda r\colon \mathbb{R}.\, \lambda p\colon float.$
$(0 \leq r \wedge (isMin\ r\ p)) \vee (r \leq 0 \wedge (isMax\ r\ p)).$

Theorem. $ToZeroRoundedModeP\colon (RoundedModeP\ ToZeroP).$

Similarly we define the rounding to infinity:

Definition. $ToInfinityP := \lambda r\colon \mathbb{R}.\, \lambda p\colon float.$
$(r \leq 0 \wedge (isMin\ r\ p)) \vee (0 \leq r \wedge (isMax\ r\ p)).$

Theorem. $ToInfinityRoundedModeP\colon (RoundedModeP\ ToInfinityP).$

While the preceeding roundings are really functions, we take advantage of having a relation to define rounding to the closest:

Definition. $Closest := \lambda r\colon \mathbb{R}.\, \lambda p\colon float.\, \mathcal{B}[p] \wedge \forall f\colon float.\, \mathcal{B}[f] \Rightarrow |p - r| \leq |f - r|.$

Theorem. $ClosestRoundedModeP\colon (RoundedModeP\ Closest).$

For the real in the middle of two successive bounded floating-point numbers there are two possible closest. So a tie-break rule is usually invoked. In our presentation, we simply accept these two points as a rounding value since uniqueness is

not required. This gives us the possibility of both proving properties that are true independently of a particular tie-break rule and investigating properties relative to a particular tie-break rule like in [29].

4 Basic Results

It is well known in any formalization that before being able to derive any interesting result, it is necessary to prove a number of elementary facts. An example of such elementary facts is the compatibility of the complement with the property of being bounded:

Theorem. *oppBounded*: $\forall x$: *float*. $\mathcal{B}[x] \Rightarrow \mathcal{B}[-x]$.

This fact is a direct consequence of our definition of the mantissa. It would not be true if we used β's complement instead of the usual sign-magnitude notation for the mantissa.

One of the first interesting result is that the difference of relatively close numbers can be done exactly with no rounding error. This property was first published by Sterbenz [34]. It has been expressed in our formalization as follows:

Theorem. *Sterbenz*: $\forall p, q$: *float*. $\mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow 1/2 * q \leq p \leq 2 * q \Rightarrow \mathcal{B}[p - q]$.

This theorem is interesting for several reasons. First of all, it contains the magic number 2. As this result is often presented and proved in binary arithmetic [13], it is not obvious if in the generic case, one has to replace 2 with β or not. For example, another property that is often used in binary arithmetic is:

Theorem. *plusUpperBound*: $\forall P$: $\mathbb{R} \rightarrow$ *float* $\rightarrow$ *Prop*. $\forall p, q, r$: *float*.
$(RoundedModeP\ P) \Rightarrow (P\ (p + q)\ r) \Rightarrow \mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow |r| \leq 2 * max(|p|, |q|)$.

In binary arithmetic this is a direct consequence of the monotony of rounding since $|p + q| \leq 2 * max(|p|, |q|)$ and $2 * max(|p|, |q|)$ is always representable in binary arithmetic. This is not the case for an arbitrary base. Take for example $\beta = 10$ with two digits of precision, rounding to the closest and $p = q = 9.9$. We have $2 * max(|p|, |q|) = 19.8$ but $(Closest\ (p + q)\ 20)$.

The Sterbenz property is also interesting by the way its proof relies on the previous definitions. The proof proceeds as follows. First of all, we restrict ourselves to the case $q \leq p \leq 2 * q$ because of the symmetry of the problem. By definition of *Fminus*, an exponent of $p - q$ is $min(e[p], e[q])$, so it is greater than or equal to $-N[b]$ since both p and q are bounded. For the mantissa, we do a case analysis on the value of $min(e[p], e[q])$. If it is $e[q]$, the initial equation can be rewritten as $0 \leq p - q \leq q$ and since $p - q$ and q have identical exponent we obtain $0 \leq n[p-q] \leq n[q]$. As q is bounded, $n[q] \leq N[b]$ allows us conclude. Similarly if $min(e[p], e[q])$=$e[p]$, we rewrite the initial equation as $0 \leq p - q \leq q \leq p$ and since $p - q$ and p have same exponent we have $0 \leq n[p - q] \leq n[p]$.

Another property that we have proved is the one concerning intervals proposed by Priest [28]. If we take two bounded positive floating-point numbers p and q and if $q - p$ can be represented exactly, then for all the floating-point numbers r inside the interval $[p, q]$, the value $r - p$ can also be represented exactly.

This is stated in our library as follows:

Theorem. *ExactMinusInterval*: $\forall P\colon \mathbb{R} \to \mathit{float} \to \mathit{Prop}.\, \forall p, q, r\colon \mathit{float}.$
$(\mathit{RoundedModeP}\ P) \Rightarrow \mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow \mathcal{B}[r] \Rightarrow 0 \leq p \leq r \leq q \Rightarrow$
$(\exists r'\colon \mathit{float}.\, \mathcal{B}[r'] \wedge r' == q - p) \Rightarrow (\exists r'\colon \mathit{float}.\, \mathcal{B}[r'] \wedge r' == r - p).$

This is a nice property but more interestingly this is the only theorem in our library that requires an inductive proof. Our proof follows the steps given by Priest. The cases where $p \leq 2 * q$ or $r \leq 2 * p$ can be proved easily using the Sterbenz property. For the other cases, we take an arbitrary r in $]2 * p, q]$ and show that if the property holds for r it holds for $(\mathit{FPred}\ r)$.

5 An Example

In order to show how the library can be used effectively, we sketch the proof that we have done to derive the correctness of a simple test program. This program is supposed to detect the radix of the arithmetic on which it is running. It was first proposed by Malcolm [23]. Here is its formulation in a Pascal-like syntax:

```
x := 1.0;
y := 1.0;
while ((x + 1.0) - x) = 1.0 do x := 2.0 * x;
while ((x + y) - x) != y do y := y + 1.0;
```

The claim is that the final value of `y` is the base of the arithmetic. Of course this program would make no sense if the computations were done exactly. It would never leave the first loop since its test is always true, and it would never enter the second loop. The proof of correctness of this program relies on two main properties. The first one insures that by increasing the mantissa of any bounded floating-point number we still get a bounded floating-point number:

Theorem. *FboundNext*: $\forall p\colon \mathit{float}.\, \mathcal{B}[p] \Rightarrow \exists q\colon \mathit{float}.\, \mathcal{B}[q] \wedge q == \langle n[p] + 1, e[p] \rangle.$

In the case of the program, we use this property with $e[p] = 0$ to justify the fact that till $\mathtt{x} \leq N[b]$, `x+1.0` is computing with no rounding error, so the test is true.

The second property is more elaborate. It uses the fact that in a binade $[\langle \beta^{\mathit{precision}-1}, e \rangle, \langle N[b], e \rangle]$ two successive floating-point numbers are separated by exactly β^e. So if we add something less than β^e to a floating-point number, we are still between this number and its successor. So the rounding is necessarily one of the two. This is expressed by the following theorem:

Theorem. *InBinade*: $\forall P\colon \mathbb{R} \to \mathit{float} \to \mathit{Prop}.\, \forall p, q, r\colon \mathit{float}.\, \forall e\colon \mathbb{Z}.\, -E[b] \leq e \Rightarrow$
$(\mathit{RoundedModeP}\ P) \Rightarrow \mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow \langle \beta^{\mathit{precision}-1}, e \rangle \leq p \leq \langle N[b], e \rangle \Rightarrow$
$0 < q < \beta^e \Rightarrow (P\ (p + q)\ r) \Rightarrow r == p \vee r == p + \beta^e$

In the case of the program we use the previous theorem only for $e = 1$. It can be rewritten as:

Theorem. *InBinade*$_1$: $\forall P\colon \mathbb{R} \to \mathit{float} \to \mathit{Prop}.\, \forall p, q, r\colon \mathit{float}.$
$(\mathit{RoundedModeP}\ P) \Rightarrow \mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow N[b] + 1 \leq p \leq \beta * N[b] \Rightarrow 0 < q < \beta \Rightarrow$
$(P\ (p + q)\ r) \Rightarrow r == p \vee r == p + \beta.$

This explains why we exit the first loop as soon as $N[b] < \mathtt{x}$. In that case the

test reduces to $0 = \texttt{1.0}$ or $\beta = \texttt{1.0}$. In a similar way, it explains why we remain in the second loop when $\texttt{y} < \beta$, the test reducing to 0 `!=` y or β `!=` y.

In order to prove the program correct, we use the possibility of annotating the program with assertions as proposed in [12]. The complete program has the following form:

```
x := 1.0;
y := 1.0;
while (x+1.0)-x = 1.0 do
  {invariant : ∃m: ℕ. 1 ≤ m ≤ β * N[b] ∧ m = x ∧ B[x]
     variant : β * N[b] − (Int_part x) for < }
    x:= 2.0 * x;
{∃m: ℕ. N[b] + 1 ≤ m ≤ β * N[b] ∧ m = x ∧ B[x] }
while (x+y)-x != y do
  {invariant : ∃m: ℕ. 1 ≤ m ≤ β ∧ m = y ∧ B[y]
     variant : β − (Int_part y) for < }
    y:= y + 1.0;
{y == β}
```

In the assertions we can refer to the variables of the program freely. For the first loop, we simply state the invariant that x represents an integer in the interval $[1, \beta{*}N[b]]$. The variant insures that at each iteration x becomes closer to $\beta{*}N[b]$. The function *Int_part* takes a real and returns its integer part. It is used to have the variant in $\mathbb{N}$. At the end of the first loop, x represents an integer in the interval $[N[b] + 1, \beta * N[b]]$. We have a similar invariant for the second loop but this time for the interval $[1, \beta]$. At the end of the program we have the expected conclusion. We can go one step further, adding an extra loop to get the precision:

```
n := 0;
x := 1.0;
while (x+1.0)-x = 1.0 do
  {invariant : ∃m: ℕ. 1 ≤ m ≤ β * N[b] ∧ m = x ∧ B[x]
     variant : β * N[b] − (Int_part x) for < }
    begin
        x:= y * x;
        n:= n + 1;
    end
{n = precision}
```

This game can be played even further. Programs like Paranoia [20], that includes Malcolm's algorithm, have been developed to check properties of floating-point arithmetics automatically.

6 Floating-Point Expansion

While computing with floating point numbers, we are usually going to accumulate rounding errors. So at the end of the computation, the result will be more or less accurate. Countless techniques exist to estimate the actual errors on the result [17]. One of the most popular methods is to use the so-called $1+\epsilon$ property.

This property just expresses that all operations are performed with a relative error of ϵ, i.e if we have an operation $\cdot$ and its equivalent with rounding $\odot$ we have the following relation:

$$\forall a, b\colon \mathit{float}.\, a \odot b = (a \cdot b) * (1 + \epsilon)$$

Given a computation, it is then possible to propagate errors and take the main term in ϵ to get an estimation of the accuracy. What is presented in this section is an orthogonal approach where one tries to give an exact account of computations while using floating-point arithmetic.

6.1 Two Sum

An interesting property of the four radix-2 IEEE implemented operations with rounding to the closest is that the error is always representable [4]. This property, independent of the radix, was already clear for the addition in [22] inspired by [27,26]. We take the usual convention that $+$ and $-$ are the exact functions, and $\oplus$ and $\ominus$ are the same operations but with rounding. This property holds:

Theorem. $\mathit{errorBoundedPlus}\colon \forall p, q, r\colon \mathit{float}.\, \mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow$
$(\mathit{Closest}\ (p+q)\ r) \Rightarrow \exists \mathit{error}\colon \mathit{float}.\, \mathit{error} == (p+q) - r \wedge \mathcal{B}[\mathit{error}]$.

In order to prove it, we rely on a basic property of rounding:

Theorem. $\mathit{RoundedModeRep}\colon \forall P\colon \mathbb{R} \to \mathit{float} \to \mathit{Prop}.\, \forall p, q\colon \mathit{float}.$
$(\mathit{RoundedModeP}\ P) \Rightarrow (P\ p\ q) \Rightarrow \exists m\colon \mathbb{Z}.\, q == \langle m, e[p] \rangle$.

This simply says that the rounding of an unbounded floating-point number can always be expressible with the same exponent, i.e by rounding we only lose bits. This means in particular that we can find a floating-point number *error* equal to $(p+q) - r$ whose exponent is either the one of p or the one of q. To prove that this number is bounded we just need to verify that its mantissa is bounded. To do this, we use the property of the rounding to the closest

$$\forall f\colon \mathit{float}.\, \mathcal{B}[f] \Rightarrow |(p+q) - r| \leq |(p+q) - f|$$

with $f = p$ and $f = q$ to get $|error| \leq |q|$ and $|error| \leq |p|$ respectively. As the exponent of *error* is the one of either p or q, we get that the *error* is bounded.

To compute effectively the error of a sum, one possibility is to use the program proposed by Knuth [22] copied here with Shewchuk's presentation [32]. It is composed of 6 operations:

$\texttt{TwoSum}_{\texttt{k}}(a, b) =$

1	$x := a \oplus b$
2	$b_v := x \ominus a$
3	$a_v := x \ominus b_v$
4	$b_r := b \ominus b_v$
5	$a_r := a \ominus a_v$
6	$\mathit{error} := a_r \oplus b_r$

There exist several proofs that this program is correct. Shewchuk gives a proof

for binary arithmetic with the extra condition that precision is greater than or equal to 3. Priest sets his proof in a general framework similar to ours but with the extra condition

$$\forall a\colon \mathit{float}.\ |a \oplus a| \leq 2|a|.$$

His proof is rather elegant as it makes use of general properties of arithmetic. It is the one we have formalized in our library. Knuth gives a more general proof in [22] since it does not have the extra condition given by Priest. Unfortunately his proof is very intricate and due to time constraint it has not yet been formalized in our library.

It is possible to compute the error of a sum with less than 6 operations, if we have some information on the operands. In particular, if we have $|a| \leq |b|$ Dekker [11] proposes the following 3 operations:

$\texttt{TwoSum}_{\texttt{d}}(a, b) =$

1 $\quad x := a \oplus b$
2 $\quad a_v := x \ominus b$
3 $\quad \mathit{error} := a \ominus a_v$

As a matter of fact, we can loosen a bit more the condition in binary arithmetic and only require that the inputs have a bounded representation such that $e[a] \leq e[b]$. This proof is not usually found in the literature, so we detail how it has been obtained in our development. First of all, the only problematic situation is when $|b| < |a|$ and $e[a] \leq e[b]$. But in this case, we can just reduce the problem to $|b| < |a|$ and $e[a] = e[b]$ and by symmetry we can suppose a positive. If b is negative, a and b being of opposite sign with same exponent, their sum is exact. When the first sum is exact, the correctness is insured because we have $a_v = a$ and $\mathit{error} = 0$. So we can suppose $0 \leq b \leq a$. If $n[a] + n[b] \leq N[b]$, the sum of a and b is computed exactly. So we are left with $N[b]+1 \leq n[a]+n[b] < 2*N[b]$. In that case it is easy to show that the second operation is performed without any rounding error, i.e. $a_v = (a \oplus b) - b$. This means that $a - a_v = (a+b) - (a \oplus b)$ which is rounding exactly as we know that the quantity on the right is representable.

The condition $e[a] \leq e[b]$ was raised in [8] by an algorithm that was working on tests but that cannot be proved with the usual condition of $|a| \leq |b|$. Giving the condition would have been more difficult had we decided to hide all the equivalent floating-point numbers behind the unique canonical representant. For this reason, Knuth only proved his theorem under the condition that the *canonical representations* of the inputs verify $e[a] \leq e[b]$ [22].

6.2 Expansion

In the previous section we have seen that in floating-point arithmetic with rounding to the closest it is possible to represent exactly a sum by a pair composed of the rounded sum and the error. Expansions are a generalisation of this idea, trying to represent a multiple precision floating-point number as a list of bounded floating-point numbers.

This technique is very efficient when multiple precision is needed for just a few operations, the inputs are floating-point numbers and the output is either

a floating-point number or a boolean value. Using a conventional high radix multiple precision package such as GMP [15] would require a lot of work for converting the input from floating-point number to the internal format. On the contrary, the best asymptotic algorithms are only available with a conventional notation. As the intermediate results need more words to be stored precisely and the number of operations grows, conventional multiple precision arithmetic will turn out to be better than expansions.

To give an exact account on the definition of expansion, we first need to define the notion of *most significant bit* and *least significant bit*. The most significant bit is represented by the function:

Definition. $MSB\colon float \to \mathbb{Z} := \lambda p\colon float.\ digit(p) + e[p]$.

The characteristic property of this function is the following:

Theorem. $ltMSB\colon \forall p\colon float.\ \neg(p == 0) \Rightarrow \beta^{(MSB\ p)} \leq |p| < \beta^{(MSB\ p)+1}$.

For the least significant bit, we need an intermediate function *maxDiv* that, given a floating-point number p, returns the greatest natural number n smaller than *precision* such that β^n divides $n[p]$. With this function, we can define the least significant bit as:

Definition. $LSB\colon float \to \mathbb{Z} := \lambda p\colon float.\ maxDiv(p) + e[p]$.

One of the main properties of the least significant bit is the following:

Theorem. $LSBrep\colon \forall p, q\colon float.$
$\neg(q == 0) \Rightarrow (LSB\ p) \leq (LSB\ q) \Rightarrow \exists z\colon \mathbb{Z}.\ q == \langle z, e[p]\rangle$.

Expansions are defined as lists of bounded floating-point numbers that do not overlap. As arithmetic algorithms manipulating expansions usually need the components to be sorted, our lists are arbitrarily sorted from the smallest number to the largest one. Also, zero elements are allowed at any place in the expansion. This is done in order not to have to necessarily insert a test to zero after every elementary operation. It also simplifies the presentation of the algorithms. Using the Prolog convention to denote list, we have the following definition:

Inductive. $IsExpansion\colon (list\ float) \to Prop :=$
$\quad Nil : (IsExpansion\ [])$
$\mid\ Single : \forall p\colon float.\ \mathcal{B}[p] \Rightarrow (IsExpansion\ [p])$
$\mid\ Top_1 : \forall p\colon float.\ \forall L\colon (list\ float).$
$\quad \mathcal{B}[p] \Rightarrow p == 0 \Rightarrow (IsExpansion\ L) \Rightarrow (IsExpansion\ [p|L])$
$\mid\ Top_2 : \forall p, q\colon float.\ \forall L\colon (list\ float).$
$\quad \mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow q == 0 \Rightarrow (IsExpansion\ [p|L]) \Rightarrow (IsExpansion\ [p, q|L])$
$\mid\ Top : \forall p, q\colon float.\ \forall L\colon (list\ float).\ \mathcal{B}[p] \Rightarrow \mathcal{B}[q] \Rightarrow (IsExpansion\ [q|L]) \Rightarrow$
$\quad \neg p == 0 \Rightarrow \neg q == 0 \Rightarrow (MSB\ p) < (LSB\ q) \Rightarrow (IsExpansion\ [p, q|L])$

It is direct to associate an expansion with the value it represents by the following function:

Fixpoint. *expValue* $[L :\ (list\ float)] :\quad float :=$
  Cases. L **of**
    $[] \Longrightarrow \langle 0, -E[b]\rangle$
  $|\ [p|L_1] \Longrightarrow p\ +\ (expValue\ L_1)$
  end.

Finally, every unbounded floating-point number that has a representation with an exponent larger than $-E[b]$ has an expansion representation. It is sufficient to break its large mantissa into smaller ones. For example, if we take the number $\langle 11223344, 0\rangle$ with an arithmetic in base 10 and 2 digits of precision, a possible expansion is $[\langle 44, 0\rangle, \langle 33, 2\rangle, \langle 22, 4\rangle, \langle 11, 6\rangle]$. We can see this construction as a recursive process. $\langle 11, 6\rangle$ is the initial number rounded to zero and $[\langle 44, 0\rangle, \langle 33, 2\rangle, \langle 22, 4\rangle]$ is the expansion representing the error done by rounding to zero. Using this process we get the following theorem:

Theorem. *existExp*: $\forall p$: *float*.
  $-E[b] \leq (LSB\ p) \Rightarrow \exists L$: $(list\ float).\ (IsExpansion\ L) \wedge p == (expValue\ L)$.

A similar result could be obtained using rounding to the closest.

6.3 Adding Two Expansions

Once we have expansions, we can start writing algorithms to manipulate them. Here we present a relatively simple but not too naive way of adding expansions given in [32] and formalized using our library. This algorithm does not use any comparison. In a deeply pipelined processor, a branch prediction miss costs many clock cycles. When the number of components of the inputs is relatively small, we get better results with this algorithm compared to asymptotically faster algorithms.

To build this adder, we suppose the existence of a function `TwoSum` that takes two floating-point numbers p and q and returns a pair of floating-point numbers (h, c) such that $h == p \oplus q$ and $c == (p + q) - (p \oplus q)$. Using this basic function, we first define a function that adds a single number to an expansion:

Fixpoint. *growExp* $[p :\ float;\ L : (list\ float)] :\quad (list\ float) :=$
  Cases. L **of**
    $[] \Longrightarrow [p]$
  $|\ [q|L_1] \Longrightarrow$ `let` $(h, c) = (TwoSum\ p\ q)$ `in` $[c|(growExp\ h\ L_1)]$
  end.

It is quite direct to see that this function returns an expansion and is correct:

Theorem. *growExpIsExp*: $\forall L$: $(list\ float).\ \forall p$: $float.\ \mathcal{B}[p] \Rightarrow$
  $(IsExpansion\ L) \Rightarrow (IsExpansion\ (growExp\ p\ L))$.

Theorem. *growExpIsVal*: $\forall L$: $(list\ float).\ \forall p$: $float.\ \mathcal{B}[p] \Rightarrow$
  $(IsExpansion\ L) \Rightarrow (expValue\ (growExp\ p\ L)) == p + (expValue\ L)$.

The naive algorithm for adding two expansions is to repeatedly add all the elements of the first expansion to the second using *growExp*. In fact, because

expansions are sorted, we can do slightly better:

Fixpoint. *addExp* $[L_1, L2 : (list\ float)] : \quad (list\ float) :=$
  Cases. L_1 **of**
        $[] \Longrightarrow L_2$
    $\mid [p|L'_1] \Longrightarrow$ **Cases.** $(growExp\ p\ L_2)$ **of**
                $[] \Longrightarrow L'_1$
              $[q|L'_2] \Longrightarrow [q|(addExp\ L'_1\ L'_2)]$
               end
  end.

The recursive call can be seen as an optimised form of the naive recursive call $(addExp\ L'_1\ [q|L'_2])$. Because q is at most comparable with p, q is 'smaller' than any element of L'_1 and 'smaller' than any element of L'_2, so it appears first and unchanged by the addition. This is the key result to prove that addition returns an expansion, while the correctness is direct:

Theorem. *addExpIsExp*: $\forall L_1, L_2$: $(list\ float)$.
$(IsExpansion\ L_1) \Rightarrow (IsExpansion\ L_2) \Rightarrow (IsExpansion\ (addExp\ L_1\ L_2))$.

Theorem. *addExpIsVal*: $\forall L_1, L_2$: $(list\ float)$. $(IsExpansion\ L_1) \Rightarrow (IsExpansion\ L_2) \Rightarrow$
$(expValue\ (addExp\ L_1\ L_2)) == (expValue\ L_1)\ +\ (expValue\ L_2)$.

7 Conclusion

We hope that what we have presented in this paper shows how effectively our floating-point library can already be used to do some verification tasks. Compared to previous works on the subject, the main originality is its genericity. No base and no format are pre-supposed and rounding is defined abstractly. Other libraries such as [16,31] follow the IEEE 754 standard and are restricted to base 2. An exception is [25] where the IEEE 784 standard is formalized, so it accommodates bases 2 and 10. We believe that most of the proofs in these libraries do not rely on the actual value of the base. This is the case, for example, in [25] where one could remove the assumption on the base and rerun the proofs without any problem. The situation is somewhat different for rounding. Other libraries define rounding as one of the four usual rounding modes. We are more liberal as we only ask for some specific properties to be met by the rounding. For example, the program in Section 5 is proved correct for an arbitrary rounding mode. Also some properties have been proved for rounding to the closest independently of a particular tie-break rule.

The core library represents 10000 lines of code for 60 definitions and 400 theorems. It is freely available from `http://www-sop.inria.fr/lemma/AOC/coq`. It is difficult to compare our library with others. Libraries such as [16,31] have been intensively used for large verification works. Most probably they are more complete than ours. Still some basic results needed for reasoning about expansions can only be found in our library.

Working on this library makes us realize that proofs really depend on the domain of application. Most proofs have been done without using any induction principle and consist mainly of nested case analysis. This clearly indicates

a limit to the generic approach of provers like Coq and the need for the development of specific tools. This is especially true for the presentation of proofs. Tactic-based theorem proving is not adequate to represent proof scripts. A more declarative approach *à la* Isar [36] would be more than needed in order to be able to communicate our proofs to non-specialists of Coq.

Finally, we are aware that building a library is a never-ending process. New applications could give rise to new elementary results and a need for some global reorganization of the library. In order to get a more stable and complete core library, we plan to work further on expansions. Recent works such as [32,9] have proposed elaborated algorithms to manipulate expansions. Getting a computer-checked version of these algorithms is a challenging task.

References

1. David H. Bailey, Robert Krasny, and Richard Pelz. Multiple precision, multiple processor vortex sheet roll-up computation. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 52–56, Philadelphia, Pennsylvania, 1993.
2. Geoff Barrett. Formal Methods Applied to a Floating-Point Number System. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.
3. Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors. *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, number 1690 in LNCS, Nice, France, September 1999. Springer-Verlag.
4. Gerd Bohlender, Wolfgang Walter, Peter Kornerup, and David W. Matula. Semantics for exact floating point operations. In Peter Kornerup and David W. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 22–26, Grenoble, France, 1991. IEEE Computer Society Press.
5. C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pages 341–350, Miami, Florida, 1999.
6. William J. Cody. Static and dynamic numerical characteristics of floating point arithmetic. *IEEE Transactions on Computers*, 22(6):598–601, 1973.
7. William J. Cody, Richard Karpinski, et al. A proposed radix and word-length independent standard for floating point arithmetic. *IEEE Micro*, 4(4):86–100, 1984.
8. Marc Daumas. Multiplications of floating point expansions. In Israel Koren and Peter Kornerup, editors, *Proceedings of the 14th Symposium on Computer Arithmetic*, pages 250–257, Adelaide, Australia, 1999.
9. Marc Daumas and Claire Finot. Division of Floating-Point Expansions with an application to the computation of a determinant. *Journal of Universal Computer Science*, 5(6):323–338, 1999.
10. Marc Daumas and Philippe Langlois. Additive symmetric: the non-negative case. *Theoretical Computer Science*, 2002.
11. T. J. Dekker. A Floating-Point Technique for Extending the Available Precision. *Numerische Mathematik*, 18(03):224–242, 1971.
12. Jean-Christophe Filliâtre. Proof of Imperative Programs in Type Theory. In *International Workshop, TYPES '98, Kloster Irsee, Germany*, number 1657 in LNCS. Springer-Verlag, March 1998.

13. David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.
14. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.
15. Tobjörn Granlund. *The GNU multiple precision arithmetic library*, 2000. Version 3.1.
16. John Harrison. A Machine-Checked Theory of Floating Point Arithmetic. In Bertot et al. [3], pages 113–130.
17. Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 1996.
18. Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial: Version 6.1. Technical Report 204, INRIA, 1997.
19. V. Karamcheti, C. Li, I. Pechtchanski, and Chee Yap. A core library for robust numeric and geometric computation. In *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pages 351–359, Miami, Florida, 1999.
20. Richard Karpinski. PARANOIA: a floating-point benchmark. *Byte*, 10(2):223–235, 1985.
21. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. advances in formal methods. Kluwer Academic Publishers, 2000.
22. Donald E. Knuth. *The art of computer programming: Seminumerical Algorithms*. Addison Wesley, 1973. Second edition.
23. Michael A. Malcolm. Algorithms to reveal properties of floating-Point Arithmetic. *Communications of the ACM*, 15(11):949–951, 1972.
24. Micaela Mayero. The Three Gap Theorem: Specification and Proof in Coq. Technical Report 3848, INRIA, 1999.
25. Paul S. Miner. Defining the IEEE-854 floating-point standard in pvs. Technical Memorandum 110167, NASA, Langley Research Center, 1995.
26. Ole Møller. Note on quasi double-precision. *BIT*, 5(4):251–255, 1965.
27. Ole Møller. Quasi double-precision in floating point addition. *BIT*, 5(1):37–50, 1965.
28. Douglas M. Priest. On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate. Phd, U.C. Berkeley, 1993.
29. John F. Reiser and Donald E. Knuth. Evading the drift in floating point addition. *Information Processing Letter*, 3(3):84–87, 1975.
30. John M. Rushby, Natajaran Shankar, and Mandayam Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, volume 1102 of *LNCS*. Springer-Verlag, July 1996.
31. David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.
32. Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(03):305–363, 1997.
33. Mike J. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.
34. Pat H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.
35. David Stevenson et al. An american national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.
36. Markus Wenzel. A Generic Interpretative Approach to Readable Formal Proof Documents. In Bertot et al. [3], pages 167–184.